



# **User's Manual**

## **V1.28.01**

**Micriµm**  
For the Way Engineers Work

Micrium  
1290 Weston Road, Suite 306  
Weston, FL 33326  
USA

[www.Micrium.com](http://www.Micrium.com)

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where Micrium Press is aware of a trademark claim, the product name appears in initial capital letters, in all capital letters, or in accordance with the vendor's capitalization preference. Readers should contact the appropriate companies for more complete information on trademarks and trademark registrations. All trademarks and registered trademarks in this book are the property of their respective holders.

Copyright © 2010 by Micrium except where noted otherwise. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher; with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

The programs and code examples in this book are presented for instructional value. The programs and examples have been carefully tested, but are not guaranteed to any particular purpose. The publisher does not offer any warranties and does not guarantee the accuracy, adequacy, or completeness of any information herein and is not responsible for any errors and omissions. The publisher assumes no liability for damages resulting from the use of the information in this book or for any infringement of the intellectual property rights of third parties that would result from the use of this information.

## USER'S MANUAL VERSIONS

If you find any errors in this document, please inform us and we will make the appropriate corrections for future releases.

Manual Version	Date	By	Description
V1.23	2009/06/22	SR	Created Manual
V1.23	2009/07/05	ITJ	Updated Manual
V1.23	2009/07/13	ITJ	Released Manual V1.23
V1.24	2009/08/24	ITJ	Added CPU Timestamp Notes/Examples
V1.24	2009/12/07	ITJ	Added CPU Timestamp Frequency
V1.25	2010/01/10	ITJ	Updated CPU Timestamp Changes
V1.26	2010/04/19	ITJ	Updated Manual
V1.27	2010/06/18	ITJ	Updated Manual
V1.27a	2010/09/20	ITJ	Converted document from Word to FrameMaker
V1.28	2010/12/17	ITJ	Updated Manual
V1.28.00	2011/02/16	ITJ	Re-released V1.28 Manual as V1.28.00

# Table of Contents

<b>0-1</b>	User's Manual Versions .....	3
<b>Chapter 1</b>	Introduction .....	6
<b>1-1</b>	Portable .....	6
<b>1-2</b>	Scalable .....	6
<b>1-3</b>	Coding Standards .....	6
<b>1-4</b>	MISRA C .....	7
<b>1-5</b>	Safety Critical Certification .....	7
<b>1-6</b>	μC/CPU Limitations .....	7
<b>Chapter 2</b>	Directories and Files .....	8
<b>Chapter 3</b>	μC/CPU Processor/Compiler Port Files .....	11
<b>3-1</b>	Standard Data Types .....	11
<b>3-2</b>	CPU Words .....	11
<b>3-2-1</b>	CPU Word Sizes .....	11
<b>3-2-2</b>	CPU Word-Memory Order .....	12
<b>3-3</b>	CPU Stacks .....	12
<b>3-4</b>	CPU Critical Sections .....	12
<b>3-4-1</b>	CPU_SR_ALLOC() .....	13
<b>3-4-2</b>	CPU_CRITICAL_ENTER() .....	14
<b>3-4-3</b>	CPU_CRITICAL_EXIT() .....	16
<b>3-4-4</b>	CPU_INT_DIS() .....	17
<b>3-4-5</b>	CPU_INT_EN() .....	18

---

<b>Chapter 4</b>	<b>µC/CPU Core Library .....</b>	<b>20</b>
4-1	µC/CPU Core Library Configuration .....	20
4-2	µC/CPU Core Library Functions and Macros .....	20
4-2-1	CPU_Init()	20
4-2-2	CPU_SW_EXCEPTION()	21
4-2-3	CPU_CntLeadZeros()	23
<b>Chapter 5</b>	<b>µC/CPU Host Name .....</b>	<b>25</b>
5-1	µC/CPU Host Name Configuration .....	25
5-2	µC/CPU Host Name Functions .....	26
5-2-1	CPU_NameClr()	26
5-2-2	CPU_NameGet()	26
5-2-3	CPU_NameSet()	28
<b>Chapter 6</b>	<b>µC/CPU Timestamps .....</b>	<b>30</b>
6-1	µC/CPU Timestamps Configuration .....	31
6-2	µC/CPU Timestamps Functions .....	31
6-2-1	CPU_TS_Get32()	31
6-2-2	CPU_TS_Get64()	32
6-2-3	CPU_TS_Update()	34
6-2-4	CPU_TS_TmrInit()	35
6-2-5	CPU_TS_TmrRd()	37
6-2-6	CPU_TS_TmrFreqGet()	40
6-2-7	CPU_TS_TmrFreqSet()	41
6-2-8	CPU_TS32_to_uSec()	42
6-2-9	CPU_TS64_to_uSec()	44
<b>Chapter 7</b>	<b>µC/CPU Interrupts Disabled Time Measurement .....</b>	<b>46</b>
7-1	µC/CPU Interrupts Disabled Time Measurement Configuration .....	46
7-2	µC/CPU Interrupts Disabled Time Measurement Functions .....	47
7-2-1	CPU_IntDisMeasMaxGet()	47
7-2-2	CPU_IntDisMeasMaxCurGet()	48
7-2-3	CPU_IntDisMeasMaxCurReset()	49
<b>Appendix A</b>	<b>µC/CPU Licensing Policy .....</b>	<b>50</b>

# Chapter

# 1

## Introduction

Designed with Micrium's renowned quality, scalability and reliability, the purpose of µC/CPU is to provide a clean, organized ANSI C implementation of each processor's/ compiler's hardware-dependent.

### **1-1 PORTABLE**

µC/CPU was designed for the vast variety of embedded applications. The processor-dependent source code for µC/CPU is designed to be ported to any processor (CPU) and compiler while µC/CPU's core library source code is designed to be independent of and used with any processor/compiler.

### **1-2 SCALABLE**

The memory footprint of µC/CPU can be adjusted at compile time based on the features you need and the desired level of run-time performance.

### **1-3 CODING STANDARDS**

Coding standards have been established early in the design of µC/CPU and include:

- C coding style
- Naming convention for `#define` constants, macros, variables and functions
- Commenting
- Directory structure

---

## 1-4 MISRA C

The source code for µC/CPU follows the Motor Industry Software Reliability Association (MISRA) C Coding Standards. These standards were created by MISRA to improve the reliability and predictability of C programs in critical automotive systems. Members of the MISRA consortium include Delco Electronics, Ford Motor Company, Jaguar Cars Ltd., Lotus Engineering, Lucas Electronics, Rolls-Royce, Rover Group Ltd., and other firms and universities dedicated to improving safety and reliability in automotive electronics. Full details of this standard can be obtained directly from the MISRA web site, <http://www.misra.org.uk>.

## 1-5 SAFETY CRITICAL CERTIFICATION

µC/CPU has been designed and implemented with safety critical certification in mind. µC/CPU is intended for use in any high-reliability, safety-critical systems including avionics RTCA DO-178B and EUROCAE ED-12B, medical FDA 510(k), IEC 61508 industrial control systems, and EN-50128 rail transportation and nuclear systems.

For example, the FAA (Federal Aviation Administration) requires that all the source code for an application be available in source form and conforming to specific software standards in order to be certified for avionics systems. Since most standard library functions are provided by compiler vendors in uncertifiable binary format, µC/CPU provides its library functions in certifiable source-code format.

If your product is not safety critical, you should view the software and safety-critical standards as proof that µC/CPU is a very robust and highly-reliable software module.

## 1-6 µC/CPU LIMITATIONS

By design, we have limited some of the feature of µC/CPU:

- Support for 64-bit data not available for all CPUs

# Chapter

# 2

## Directories and Files

The distribution of µC/CPU is typically included in a ZIP file called: `Micrium_uC-CPU-Vxxy.zip`. (Note: The ZIP file name might also include customer names, invoice numbers, and file creation date.) The ZIP file contains all the source code and documentation for µC/CPU organized in a directory structure according to “AN-2002, µC/OS-II Directory Structure.” Specifically, the files may be found in the following directories:

### **\Micrium\Software\uC-CPU**

This is the main directory for µC/CPU and contains generic, processor-independent source code including:

#### `cpu_def.h`

This file declares `#define` constants used to configure processor/compiler-specific CPU word sizes, endianness word order, critical section methods, and other processor configuration.

#### `cpu_core.c` and `cpu_core.h`

These files contain source code that implements µC/CPU features such as host name allocation, timestamps, time measurements, and counting lead zeros.

### **\Micrium\Software\uC-CPU\Doc**

This directory contains all µC/CPU documentation files.

### **\Micrium\Software\uC-CPU\Cfg\Template**

This directory contains a template file, `cpu_cfg.h`, which includes configuration for µC/CPU features such as host name allocation, timestamps, time measurements, and assembly optimization. Your application must include a `cpu_cfg.h` configuration file with application-specific configuration settings.

### **\Micrium\Software\uC-CPU\BSP\Template**

This directory contains a template file, `cpu_bsp.c`, which includes function templates for the board-specific (BSP) code required if certain µC/CPU features such as timestamp time measurements and assembly optimization are enabled. Your application must include code for all BSP functions enabled in `cpu_cfg.h`.

### **\Micrium\Software\<CPU Type>\<Compiler>**

µC/CPU also contains additional sub-directories specific to each processor/compiler combination organized as follows:

#### **cpu.h**

This file contains µC/CPU configuration specific to the processor (CPU Type) and compiler (Compiler), such as data type definitions, processor address and data word sizes, endianness word order, and critical section macros. See Chapter 3, “µC/CPU Processor/Compiler Port Files” on page 11 for more details.

#### **cpu\_a.asm or cpu\_a.s**

These (optional) files contain assembly code to enable/disable interrupts, implement critical section methods, and any other processor-specific code not already defined or implemented in the processor’s `cpu.h` (or `cpu.c`).

#### **cpu.c**

This (optional) file contains C and/or assembly code to implement processor-specific code not already defined or implemented in the processor’s `cpu.h` (or `cpu_a.asm`).

#### **\Template\cpu.h and cpu\_a.asm**

These template µC/CPU configuration files include example configurations for a generic processor/compiler.

An example of ARM-specific CPU processor files is shown in Figure 2-1:

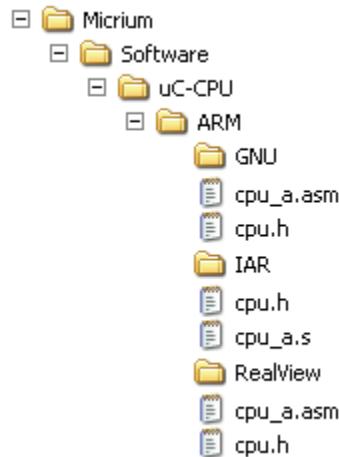


Figure 2-1 **µC/CPU ARM CPU Directories and Files Example**

Application files which intend to make use of µC/CPU macros or functions should `#include` the desired µC/CPU header files. In addition, applications are required to configure µC/CPU features in application-specific configuration file, `cpu_cfg.h`.

# Chapter 3

## $\mu$ C/CPU Processor/Compiler Port Files

$\mu$ C/CPU contains configuration specific to each processor and compiler, such as standard data type definitions, processor address and data word sizes, endianness word order, critical section macros, and possibly other functions and macros. These are defined in each specific processor/compiler subdirectory's `cpu.h`.

### **3-1 STANDARD DATA TYPES**

$\mu$ C/CPU ports define standard data types such as `CPU_CHAR`, `CPU_BOOLEAN`, `CPU_INT08U`, `CPU_INT16S`, `CPU_FP32`, etc. These data types are used in Micrium applications, and may be used in your applications, to facilitate portability independent of and between processors/compilers. Most  $\mu$ C/CPU processor/compiler port files minimally support 32-bit data types, but may optionally support 64-bit (or greater) data types.

In addition, several regularly-used function pointer data types are defined.

### **3-2 CPU WORDS**

#### **3-2-1 CPU WORD SIZES**

$\mu$ C/CPU ports include word size configuration such as `CPU_CFG_ADDR_SIZE` and `CPU_CFG_DATA_SIZE`, configured via `CPU_WORD_SIZE_08`, `CPU_WORD_SIZE_16`, and `CPU_WORD_SIZE_32`.

In addition, the following CPU word sizes are also defined based on the configured sizes of `CPU_CFG_ADDR_SIZE` and `CPU_CFG_DATA_SIZE` : `CPU_ADDR`, `CPU_DATA`, `CPU_ALIGN`, and `CPU_SIZE_T`.

### **3-2-2 CPU WORD-MEMORY ORDER**

μC/CPU ports configure `CPU_CFG_ENDIAN_TYPE` to indicate the processor's word-memory order endianness. `CPU_ENDIAN_TYPE_LITTLE` indicates that a CPU stores/reads data words in memory with the most significant octets at lower memory addresses (and the least significant octets at higher memory addresses) while a `CPU_ENDIAN_TYPE_BIG` CPU stores/reads data words in memory with the most significant octets at higher memory addresses (and the least significant octets at lower memory addresses).

### **3-3 CPU STACKS**

μC/CPU ports configure `CPU_CFG_STK_GROWTH` to indicate the direction in memory a CPU updates its stack pointers after pushing data onto its stacks. `CPU_STK_GROWTH_HI_TO_LO` indicates that a CPU decrements its stack pointers to the next lower memory address after data is pushed onto a CPU stack while a `CPU_STK_GROWTH_LO_TO_HI` CPU increments its stack pointers to the next higher memory address after data is pushed.

In addition, each μC/CPU processor port defines a `CPU_STK` data type to the CPU's stack word size.

### **3-4 CPU CRITICAL SECTIONS**

μC/CPU ports include CPU critical section configuration `CPU_CFG_CRITICAL_METHOD` that indicates how a CPU disables/re-enables interrupts when entering/exiting critical, protected sections:

`CPU_CRITICAL_METHOD_INT_DIS_EN` merely disables/enables interrupts on critical section enter/exit. This is not a preferred method since it does not support multiple levels of interrupts. However, with some processors/compilers, this is the only available method.

`CPU_CRITICAL_METHOD_STATUS_STK` pushes/pops interrupt status onto stack before disabling/re-enabling interrupts. This is one preferred method since it supports multiple levels of interrupts. However, this method assumes that the compiler provides C-level and/or assembly-level functionality for pushing/saving the interrupt status onto a local stack, disabling interrupts, and popping/restoring the interrupt status from the local stack.

`CPU_CRITICAL_METHOD_STATUS_LOCAL` saves/restores interrupt status to a local variable before disabling/re-enabling interrupts. This also is a preferred method since it supports

multiple levels of interrupts. However, this method assumes that the compiler provides C-level and/or assembly-level functionality for saving the interrupt status to a local variable, disabling interrupts, and restoring the interrupt status from the local variable.

Each μC/CPU processor port implements critical section macros with calls to interrupt disable/enable macros. Applications should only use the critical section macros (see section 3-4-2 “CPU\_CRITICAL\_ENTER()” on page 14 and section 3-4-3 “CPU\_CRITICAL\_EXIT()” on page 16) since interrupt disable/enable macros (see section 3-4-4 “CPU\_INT\_DIS()” on page 17 and section 3-4-5 “CPU\_INT\_EN()” on page 18) are intended for use only by core μC/CPU functions.

Each μC/CPU processor port may define its interrupt disable/enable macros with inline-assembly directly in `cpu.h`, or calls to C functions defined in `cpu.c`, or calls to assembly subroutines defined in `cpu_a.asm` (or `cpu_a.s`). The specific implementation should be based on the processor port’s configured CPU critical section method.

In addition, each μC/CPU processor port defines an appropriately-sized `CPU_SR` data type large enough to completely store the processor’s/compiler’s status word. `CPU_CRITICAL_METHOD_STATUS_LOCAL` method requires each function that calls critical section macros or interrupt disable/enable macros to declare local variable `cpu_sr` of type `CPU_SR`, which should be declared via the `CPU_SR_ALLOC()` macro (see section 3-4-1).

### **3-4-1 CPU\_SR\_ALLOC()**

Allocates CPU status register word as local variable `cpu_sr`, when necessary, for use with critical section macros.

#### **FILES**

Each specific processor’s/compiler’s `cpu.h`

#### **PROTOTYPE**

```
CPU_SR_ALLOC();
```

## ARGUMENTS

None.

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

None.

## NOTES / WARNINGS

`CPU_SR_ALLOC()` *must* be called immediately after the last local variable declaration in a function but before any code statements.

## EXAMPLE USAGE

```
CPU_BOOLEAN  ts_init;
CPU_TS        ts_cur;
CPU_SR_ALLOC();           /* Declared immediately after all local variables ... */
                         /* ... but before any code statements.          */
ts_init = DEF_YES;
ts_cur  = CPU_TS_TmrRd();
```

### 3-4-2 `CPU_CRITICAL_ENTER()`

Enters critical sections, disabling interrupts.

## FILES

Each specific processor's/compiler's `cpu.h`

## PROTOTYPE

```
CPU_CRITICAL_ENTER();
```

## ARGUMENTS

None.

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

None.

## NOTES / WARNINGS

`CPU_CRITICAL_ENTER()` / `CPU_CRITICAL_EXIT()` should be used to protect critical sections of code from interrupted or concurrent access when no other protection mechanisms are available or appropriate. For example, system code that must be re-entrant but without use of a software lock should protect the code using CPU critical sections.

Since interrupts are disabled upon calling `CPU_CRITICAL_ENTER()` and are not re-enabled until after calling `CPU_CRITICAL_EXIT()`, interrupt and operating system context switching are postponed while all critical sections have not completely exited.

Critical sections can be nested any number of times as long as `CPU_CFG_CRITICAL_METHOD` is not configured as `CPU_CRITICAL_METHOD_INT_DIS_EN`, which would re-enable interrupts upon the first call to `CPU_CRITICAL_EXIT()`, not the last call.

`CPU_CRITICAL_ENTER()` *should/must* always call `CPU_CRITICAL_EXIT()` once critical section protection is no longer needed.

## EXAMPLE USAGE

```
CPU_SR_ALLOC();

CPU_CRITICAL_ENTER();
:
:           /* Code protected by critical sections ... */
:           /* ... from interrupts or concurrent access. */
:
CPU_CRITICAL_EXIT();
```

### **3-4-3 CPU\_CRITICAL\_EXIT()**

Exits critical sections, restoring previous interrupt status and/or enabling interrupts.

#### **FILES**

Each specific processor's/compiler's `cpu.h`

#### **PROTOTYPE**

```
CPU_CRITICAL_EXIT();
```

#### **ARGUMENTS**

None.

#### **RETURNED VALUE**

None.

#### **REQUIRED CONFIGURATION**

None.

#### **NOTES / WARNINGS**

`CPU_CRITICAL_ENTER()` / `CPU_CRITICAL_EXIT()` should be used to protect critical sections of code from interrupted or concurrent access when no other protection mechanisms are available or appropriate. For example, system code that must be re-entrant but without use of a software lock should protect the code using CPU critical sections.

Since interrupts are disabled upon calling `CPU_CRITICAL_ENTER()` and are not re-enabled until after calling `CPU_CRITICAL_EXIT()`, interrupt and operating system context switching are postponed while all critical sections have not completely exited.

Critical sections can be nested any number of times as long as `CPU_CFG_CRITICAL_METHOD` is not configured as `CPU_CRITICAL_METHOD_INT_DIS_EN`, which would re-enable interrupts upon the first call to `CPU_CRITICAL_EXIT()`, not the last call.

`CPU_CRITICAL_EXIT()` *must* always call `CPU_CRITICAL_ENTER()` at the start of critical section protection.

## EXAMPLE USAGE

```
CPU_SR_ALLOC();  
  
CPU_CRITICAL_ENTER();  
:  
:  
:           /* Code protected by critical sections ... */  
:           /* ... from interrupts or concurrent access. */  
:  
CPU_CRITICAL_EXIT();
```

### **3-4-4 CPU\_INT\_DIS()**

Saves current interrupt status, if processor/compiler capable, and then disables interrupts.

## FILES

Each specific processor's/compiler's `cpu.h`

## PROTOTYPE

```
CPU_INT_DIS();
```

## ARGUMENTS

None.

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

None.

## NOTES / WARNINGS

`CPU_INT_DIS()` should be defined based on the processor port's configured CPU critical section method, `CPU_CFG_CRITICAL_METHOD`; and may be defined with inline-assembly directly in `cpu.h`, or with calls to C functions defined in `cpu.c`, or calls to assembly subroutines defined in `cpu_a.asm` (or `cpu_a.s`). See also section 3-4.

## EXAMPLE TEMPLATES

The following example templates assume corresponding functions are defined in either `cpu.c` or `cpu_a.asm`:

```
#if      (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_INT_DIS_EN)
        /* Disable interrupts. */
#define CPU_INT_DIS()  do { CPU_IntDis(); } while (0)
#endif

#if      (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_STK)
        /* Push      CPU status & disable interrupts. */
#define CPU_INT_DIS()  do { CPU_SR_Push(); } while (0)
#endif

#if      (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_LOCAL)
        /* Save      CPU status & disable interrupts. */
#define CPU_INT_DIS()  do { cpu_sr = CPU_SR_Save(); } while (0)
#endif
```

## 3-4-5 `CPU_INT_EN()`

Restores previous interrupt status and/or enables interrupts.

## FILES

Each specific processor's/compiler's `cpu.h`

## PROTOTYPE

```
CPU_INT_EN();
```

## ARGUMENTS

None.

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

None.

## NOTES / WARNINGS

`CPU_INT_EN()` should be defined based on the processor port's configured CPU critical section method, `CPU_CFG_CRITICAL_METHOD`; and may be defined with inline-assembly directly in `cpu.h`, or with calls to C functions defined in `cpu.c`, or calls to assembly subroutines defined in `cpu_a.asm` (or `cpu_a.s`). See also section 3-4.

## EXAMPLE TEMPLATES

The following example templates assume corresponding functions are defined in either `cpu.c` or `cpu_a.asm`:

```
#if      (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_INT_DIS_EN)
        /* Enable  interrupts. */
#define  CPU_INT_EN()  do { CPU_IntEn(); } while (0)
#endif

#if      (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_STK)
        /* Pop      CPU status. */
#define  CPU_INT_EN()  do { CPU_SR_Pop(); } while (0)
#endif

#if      (CPU_CFG_CRITICAL_METHOD == CPU_CRITICAL_METHOD_STATUS_LOCAL)
        /* Restore CPU status. */
#define  CPU_INT_EN()  do { CPU_SR_Restore(cpu_sr); } while (0)
#endif
```

# Chapter

# 4

## µC/CPU Core Library

µC/CPU core library functions initialize µC/CPU, handle software exceptions, and include features such as counting the lead zeros in a word. These features are configured in `cpu_cfg.h` and defined in `cpu_core.c`.

### 4-1 µC/CPU CORE LIBRARY CONFIGURATION

The following core µC/CPU configurations must be configured in `cpu_cfg.h`:

<code>CPU_CFG_LEAD_ZEROS_ASM_PRESENT</code>	Implements counting lead zeros functionality in assembly (see section 4-2-3). This feature is enabled if the macro is <code>#define'd</code> in <code>cpu_cfg.h</code> (or <code>cpu.h</code> ).
---	--

### 4-2 µC/CPU CORE LIBRARY FUNCTIONS AND MACROS

#### 4-2-1 `CPU_Init()`

Initializes the core CPU module.

#### FILES

`cpu_core.h`/`cpu_core.c`

#### PROTOTYPE

```
void CPU_Init (void);
```

## ARGUMENTS

None.

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

None.

## NOTES / WARNINGS

`CPU_Init()` must be called by application code prior to calling any other core CPU functions:

- CPU host name
- CPU timestamps
- CPU interrupts disabled time measurements

## 4-2-2 `CPU_SW_EXCEPTION()`

Traps an unrecoverable software exception.

## FILES

`cpu_core.h`

## PROTOTYPE

```
CPU_SW_EXCEPTION();
```

## ARGUMENTS

<code>err_rtn_val</code>	Error type and/or value of the calling function to return.
--------------------------	--

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

None.

## NOTES / WARNINGS

Deadlocks the current code execution—whether multi-tasked/-processed/-threaded or single-threaded—when the current code execution cannot gracefully recover or report a fault or exception condition.

## EXAMPLE USAGE

```
void Fnct (CPU_ERR *p_err)
{
    if (p_err == (CPU_ERR *)0) { /* If 'p_err' NULL, cannot return error ... */
        CPU_SW_EXCEPTION();      /* ... so trap invalid argument exception. */
    }
    ...
}
```

## DEVELOPER-IMPLEMENTED EXAMPLES

`CPU_SW_EXCEPTION()` may be developer-implemented to output and/or handle any error or exception conditions; but since `CPU_SW_EXCEPTION()` is intended to trap unrecoverable software conditions, it is recommended that developer-implemented versions prevent execution of any code following calls to `CPU_SW_EXCEPTION()` by deadlocking the code.

```
#define CPU_SW_EXCEPTION(err_rtn_val)      \
    do {                                     \
        Log(__FILE__, __LINE__); \
        CPU_SW_Exception();      /* SHOULD deadlock to prevent further code execution. */ \
    } while (0)
```

Listing 4-1 **Developer-implemented** `CPU_SW_EXCEPTION()` **with deadlock**

However, if execution of code following calls to `CPU_SW_EXCEPTION()` is required (e.g. for automated testing); it is recommended that the last statement in developer-implemented versions be to return from the current function to prevent possible software exceptions in the current function from triggering CPU and/or hardware exceptions. (Note that `err_rtn_val` in the return statement *must not* be enclosed in parentheses. This allows `CPU_SW_EXCEPTION()` to return from functions that return `void`, i.e. no return type or value.)

```
#define CPU_SW_EXCEPTION(err_rtn_val)    \
    do {                                \
        Log(__FILE__, __LINE__); \
        return err_rtn_val;    /* MUST NOT enclose 'err_rtn_val' in parentheses. */ \
    } while (0)
```

Listing 4-2 **Developer-implemented** `CPU_SW_EXCEPTION()` **with return**

### 4-2-3 `CPU_CntLeadZeros()`

Counts the number of contiguous, most-significant, leading zero bits in a data value.

#### FILES

`cpu_core.h/cpu_core.c` / Specific CPU/compiler `cpu_a.asm`

#### PROTOTYPE

```
CPU_DATA CPU_CntLeadZeros (CPU_DATA val);
```

#### ARGUMENTS

`val` Data value to count leading zero bits.

#### RETURNED VALUE

Maximum interrupts disabled time (in CPU timestamp timer counts) before resetting.

## **REQUIRED CONFIGURATION**

`CPU_CntLeadZeros()` available and implemented in `cpu_core.c` if `CPU_CFG_LEAD_ZEROS_ASM_PRESENT` is not `#define'd` in `cpu_cfg.h` (or `cpu.h`), but should be implemented in `cpu_a.asm` (or `cpu_a.s`) if `CPU_CFG_LEAD_ZEROS_ASM_PRESENT` is `#define'd` in `cpu_cfg.h` (or `cpu.h`). See section 4-1.

## **NOTES / WARNINGS**

None.

## **EXAMPLE USAGE**

```
CPU_DATA val;
CPU_DATA nbr_lead_zeros;

val           = 0x0643A718;
nbr_lead_zeros = CPU_CntLeadZeros(val);
```

# Chapter 5

## µC/CPU Host Name

µC/CPU host name feature allows a target host to configure a name for itself. This may be used to uniquely identify the target in a system or network of inter-connected hosts. The CPU host name feature is available only if `CPU_CFG_NAME_EN` is `DEF_ENABLED` in `cpu_cfg.h`.

### **5-1 µC/CPU HOST NAME CONFIGURATION**

The following µC/CPU host name configurations must be configured in `cpu_cfg.h` :

<code>CPU_CFG_NAME_EN</code>	Includes code to set and get a configured CPU host name. This feature may be configured to either <code>DEF_DISABLED</code> or <code>DEF_ENABLED</code> .
<code>CPU_CFG_NAME_SIZE</code>	Configures the maximum CPU name size (in number of ASCII characters, including the terminating <code>NULL</code> character).

## 5-2 $\mu$ C/CPU HOST NAME FUNCTIONS

### 5-2-1 CPU\_NameClr()

Clears the CPU host name.

#### FILES

cpu\_core.h/cpu\_core.c

#### PROTOTYPE

```
void CPU_NameClr (void);
```

#### ARGUMENTS

None.

#### RETURNED VALUE

None.

#### REQUIRED CONFIGURATION

Available only if CPU\_CFG\_NAME\_EN is DEF\_ENABLED in cpu\_cfg.h (see section 5-1).

#### NOTES / WARNINGS

CPU\_Init() must be called by application code prior to calling any other core CPU functions:

### 5-2-2 CPU\_NameGet()

Gets the CPU host name.

#### FILES

cpu\_core.h/cpu\_core.c

## PROTOTYPE

```
void CPU_NameGet (CPU_CHAR *p_name,  
                  CPU_ERR *p_err);
```

## ARGUMENTS

**p\_name** Pointer to an ASCII character array that will receive the return CPU host name ASCII string from this function.

**p\_err** Pointer to variable that will receive the return error code from this function:

CPU\_ERR\_NONE  
CPU\_ERR\_NULL\_PTR

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

Available only if CPU\_CFG\_NAME\_EN is DEF\_ENABLED in `cpu_cfg.h` (see section 5-1).

## NOTES / WARNINGS

The size of the ASCII character array that will receive the return CPU host name ASCII string must be greater than or equal to the current CPU host name's ASCII string size including the terminating NULL character; and should be greater than or equal to CPU\_CFG\_NAME\_SIZE.

---

## EXAMPLE USAGE

```
CPU_CHAR *p_name;
CPU_ERR   err;

CPU_NameGet(p_name, &err); /* Get CPU host name. */

if (err == CPU_ERR_NONE) {
    printf("CPU Host Name = %s", p_name);
} else {
    printf("COULD NOT GET CPU HOST NAME.");
}
```

### 5-2-3 CPU\_NameSet()

Sets the CPU host name.

#### FILES

cpu\_core.h/cpu\_core.c

#### PROTOTYPE

```
void CPU_NameSet (const CPU_CHAR *p_name,
                  CPU_ERR   *p_err);
```

#### ARGUMENTS

p\_name      Pointer to an ASCII character string with CPU host name to set.

p\_err      Pointer to variable that will receive the return error code from this function:

CPU\_ERR\_NONE  
CPU\_ERR\_NULL\_PTR  
CPU\_ERR\_NAME\_SIZE

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

Available only if `CPU_CFG_NAME_EN` is `DEF_ENABLED` in `cpu_cfg.h` (see section 5-1).

## NOTES / WARNINGS

`p_name`'s ASCII string size, including the terminating `NULL` character, *must* be less than or equal to `CPU_CFG_NAME_SIZE`.

## EXAMPLE USAGE

```
CPU_CHAR *p_name;
CPU_ERR   err;

p_name = "CPU Host Target";

CPU_NameSet(p_name, &err); /* Set CPU host name. */

if (err != CPU_ERR_NONE) {
    printf("COULD NOT SET CPU HOST NAME.");
}
```

# Chapter 6

## $\mu$ C/CPU Timestamps

$\mu$ C/CPU timestamps emulate a real-time 32- or 64-bit timer using any size hardware (or software) timer. If the hardware (or software) timer used has the same (or greater) number of bits as the 32- or 64-bit CPU timestamps, then calls to `CPU_TS_Get()` functions return the timer value directly with no additional calculation overhead. But if the timer has less bits than the 32- or 64-bit CPU timestamps, `CPU_TS_Update()` must be called periodically by an application-/developer-defined function (see section 6-2-3) to accumulate timer counts into the 32- or 64-bit CPU timestamps. An application can then use CPU timestamps either as raw timer counts or converted to microseconds (see section 6-2-8 and section 6-2-9).

Note that if either the CPU timestamp feature or the interrupts disable time measurement feature is enabled (see section 6-1 and section 7-1), then the application/developer must provide CPU timestamp timer functions (see section 6-2-4 “`CPU_TS_TmrInit()`” on page 35 and section 6-2-5 “`CPU_TS_TmrRd()`” on page 37). In addition, the CPU timestamp timer word size must be appropriately configured via `CPU_CFG_TS_TMR_SIZE` in `cpu_cfg.h`:

<code>CPU_WORD_SIZE_08</code>	8-bit word size
<code>CPU_WORD_SIZE_16</code>	16-bit word size
<code>CPU_WORD_SIZE_32</code>	32-bit word size
<code>CPU_WORD_SIZE_64</code>	64-bit word size

This configures the size of the `CPU_TS_TMR` data type (see section 6-2-5). Since the CPU timestamp timer must not have less bits than the `CPU_TS_TMR` data type; `CPU_CFG_TS_TMR_SIZE` must be configured so that all bits in `CPU_TS_TMR` data type are significant. In other words, if the size of the CPU timestamp timer is not a binary-multiple of 8-bit octets (e.g. 20-bits or even 24-bits), then the next lower, binary-multiple octet word size should be configured (e.g. to 16-bits). However, the minimum supported word size for CPU timestamp timers is 8-bits.

## **6-1 µC/CPU TIMESTAMPS CONFIGURATION**

The following µC/CPU timestamps configurations must be configured in `cpu_cfg.h`:

<code>CPU_CFG_TS_32_EN</code>	Includes 32-bit CPU timestamp functionality (see section 6-2-1). This feature may be configured to either <code>DEF_DISABLED</code> or <code>DEF_ENABLED</code> .
<code>CPU_CFG_TS_64_EN</code>	Includes 64-bit CPU timestamp functionality (see section 6-2-2). This feature may be configured to either <code>DEF_DISABLED</code> or <code>DEF_ENABLED</code> .
<code>CPU_CFG_TS_TMR_SIZE</code>	Configures the CPU timestamp's hardware or software timer word size (see Chapter 6, on page 30).

## **6-2 µC/CPU TIMESTAMPS FUNCTIONS**

### **6-2-1 `CPU_TS_Get32()`**

Gets current 32-bit CPU timestamp.

#### **FILES**

`cpu_core.h`/`cpu_core.c`

#### **PROTOTYPE**

```
CPU_TS32 CPU_TS_Get32 (void);
```

#### **ARGUMENTS**

None.

#### **RETURNED VALUE**

None.

## REQUIRED CONFIGURATION

Available only if `CPU_CFG_TS_32_EN` is `DEF_ENABLED` in `cpu_cfg.h` (see section 6-1).

## NOTES / WARNINGS

The amount of time measured by CPU timestamps is calculated by either of the following equations:

$$\text{Time measured} = \text{Number timer counts} * \text{Timer period}$$

where

Number timer counts	Number of timer counts measured
Timer period	Timer's period in some units of (fractional) seconds
Time measured	Amount of time measured, in same units of (fractional) seconds as the Timer period

$$\text{Time measured} = \text{Number timer counts} / \text{Timer frequency}$$

where

Number timer counts	Number of timer counts measured
Timer frequency	Timer's frequency in some units of counts per second
Time measured	Amount of time measured, in seconds

## EXAMPLE USAGE

```
CPU_TS32 ts32;  
  
ts32 = CPU_TS_Get32(); /* Get current 32-bit CPU timestamp. */
```

### 6-2-2 `CPU_TS_Get64()`

Gets current 64-bit CPU timestamp.

## FILES

`cpu_core.h`/`cpu_core.c`

---

## PROTOTYPE

```
CPU_TS32 CPU_TS_Get64 (void);
```

## ARGUMENTS

None.

## RETURNED VALUE

None.

## REQUIRED CONFIGURATION

Available only if `CPU_CFG_TS_64_EN` is `DEF_ENABLED` in `cpu_cfg.h` (see section 6-1).

## NOTES / WARNINGS

The amount of time measured by CPU timestamps is calculated by either of the following equations:

$$\text{Time measured} = \text{Number timer counts} * \text{Timer period}$$

where

Number timer counts	Number of timer counts measured
Timer period	Timer's period in some units of (fractional) seconds
Time measured	Amount of time measured, in same units of (fractional) seconds as the Timer period

$$\text{Time measured} = \text{Number timer counts} / \text{Timer frequency}$$

where

Number timer counts	Number of timer counts measured
Timer frequency	Timer's frequency in some units of counts per second
Time measured	Amount of time measured, in seconds

## EXAMPLE USAGE

```
CPU_TS64 ts64;  
  
ts64 = CPU_TS_Get64(); /* Get current 64-bit CPU timestamp. */
```

### 6-2-3 CPU\_TS\_Update()

Updates current 32- and 64-bit CPU timestamps.

#### FILES

cpu\_core.h/cpu\_core.c

#### PROTOTYPE

```
void CPU_TS_Update (void);
```

#### ARGUMENTS

None.

#### RETURNED VALUE

None.

#### REQUIRED CONFIGURATION

Available only if either CPU\_CFG\_TS\_32\_EN or CPU\_CFG\_TS\_64\_EN is DEF\_ENABLED in cpu\_cfg.h (see section 6-1).

#### NOTES / WARNINGS

CPU timestamps *must* be updated periodically by some application (or BSP) time handler in order to adequately maintain the CPU timestamps' time and *must* be updated more frequently than the CPU timestamp timer overflows; otherwise, CPU timestamps will lose time.

## EXAMPLE USAGE

```
void AppPeriodicTimeHandler (void)
{
    :
    CPU_TS_Update(); /* Update current CPU timestamps. */
    :
}
```

### 6-2-4 CPU\_TS\_TmrInit()

Application-defined function to initialize and start the CPU timestamp's (hardware or software) timer.

#### FILES

cpu\_core.h / Application's cpu\_bsp.c

#### PROTOTYPE

```
void CPU_TS_TmrInit (void);
```

#### ARGUMENTS

None.

#### RETURNED VALUE

None.

## REQUIRED CONFIGURATION

`CPU_TS_TmrInit()` is an application/BSP function that *must* be defined by the developer if either of the following CPU features is enabled in `cpu_cfg.h`:

- CPU timestamps when either `CPU_CFG_TS_32_EN` or `CPU_CFG_TS_64_EN` is `DEF_ENABLED` (see section 6-1)
- CPU interrupts disabled time measurements when `CPU_CFG_INT_DIS_MEAS_EN` is `#define'd` (see section 7-1)

## NOTES / WARNINGS

CPU timestamp timer count values must be returned via word-size-configurable `CPU_TS_TMR` data type. If timer has more bits, truncate timer values' higher-order bits greater than the configured `CPU_TS_TMR` timestamp timer data type word size. However, since the timer must not have less bits than the configured `CPU_TS_TMR` timestamp timer data type word size; `CPU_CFG_TS_TMR_SIZE` must be configured so that all bits in `CPU_TS_TMR` data type are significant. In other words, if timer size is not a binary-multiple of 8-bit octets (e.g. 20-bits or even 24-bits), then the next lower, binary-multiple octet word size should be configured (e.g. to 16-bits). However, the minimum supported word size for CPU timestamp timers is 8-bits.

CPU timestamp timer should be an 'up' counter whose values increase with each time count. If timer is a 'down' counter whose values decrease with each time count, then the returned timer value must be ones-complemented.

When applicable, CPU timestamp timer period should be less than the typical measured time but must be less than the maximum measured time; otherwise, timer resolution inadequate to measure desired times.

## EXAMPLE TEMPLATE

```
void CPU_TS_TmrInit (void)
{
    /* Insert code to initialize/start CPU timestamp timer. */ ;
}
```

## 6-2-5 CPU\_TS\_TmrRd()

Application-defined function to get current CPU timestamp timer count.

### FILES

cpu\_core.h / Application's cpu\_bsp.c

### PROTOTYPE

```
CPU_TS_TMR CPU_TS_TmrRd (void);
```

### ARGUMENTS

None.

### RETURNED VALUE

CPU timestamp timer count value.

### REQUIRED CONFIGURATION

CPU\_TS\_TmrRd() is an application/BSP function that *must* be defined by the developer if either of the following CPU features is enabled in cpu\_cfg.h :

- CPU timestamps when either CPU\_CFG\_TS\_32\_EN or CPU\_CFG\_TS\_64\_EN is `DEF_ENABLED` (see section 6-1)
- CPU interrupts disabled time measurements when CPU\_CFG\_INT\_DIS\_MEAS\_EN is `#define'd` (see section 7-1)

### NOTES / WARNINGS

CPU timestamp timer count values must be returned via word-size-configurable CPU\_TS\_TMR data type. If timer has more bits, truncate timer values' higher-order bits greater than the configured CPU\_TS\_TMR timestamp timer data type word size. However, since the timer must not have less bits than the configured CPU\_TS\_TMR timestamp timer data type word size; CPU\_CFG\_TS\_TMR\_SIZE must be configured so that all bits in CPU\_TS\_TMR data type are significant. In other words, if timer size is not a binary-multiple of 8-bit octets (e.g.

---

20-bits or even 24-bits), then the next lower, binary-multiple octet word size should be configured (e.g. to 16-bits). However, the minimum supported word size for CPU timestamp timers is 8-bits.

CPU timestamp timer should be an ‘up’ counter whose values increase with each time count. If timer is a ‘down’ counter whose values decrease with each time count, then the returned timer value must be ones-complemented.

When applicable, CPU timestamp timer period should be less than the typical measured time but must be less than the maximum measured time; otherwise, timer resolution inadequate to measure desired times.

## **EXAMPLE TEMPLATE**

```
CPU_TS_TMR  CPU_TS_TmrRd (void)
{
    CPU_TS_TMR  ts_tmr_cnts;

    ...
    ts_tmr_cnts = /* Insert code to get/return CPU timestamp timer counts. */ ;
    ...

    return (ts_tmr_cnts);
}
```

## **16-BIT UP TIMER EXAMPLE**

```
CPU_TS_TMR  CPU_TS_TmrRd (void)
{
    CPU_TS_TMR  ts_tmr_cnts; /* sizeof(CPU_TS_TMR) = 16 bits */

    ts_tmr_cnts = /* Insert code to read 16-bit up timer value. */ ;

    return (ts_tmr_cnts);
}
```

---

## 16-BIT DOWN TIMER EXAMPLE

```
CPU_TS_TMR CPU_TS_TmrRd (void)
{
    CPU_INT16U tmr_val;
    CPU_TS_TMR ts_tmr_cnts; /* sizeof(CPU_TS_TMR) = 16 bits */

    tmr_val = /* Insert code to read 16-bit down timer value. */ ;
    ts_tmr_cnts = ~tmr_val; /* Ones-complement 16-bit down timer value. */

    return (ts_tmr_cnts);
}
```

## 32-BIT UP TIMER EXAMPLE

```
CPU_TS_TMR CPU_TS_TmrRd (void)
{
    CPU_TS_TMR ts_tmr_cnts; /* sizeof(CPU_TS_TMR) = 32 bits */

    ts_tmr_cnts = /* Insert code to read 32-bit up timer value. */ ;

    return (ts_tmr_cnts);
}
```

## 48-BIT DOWN TIMER EXAMPLE

```
CPU_TS_TMR CPU_TS_TmrRd (void)
{
    CPU_INT64U tmr_val;
    CPU_TS_TMR ts_tmr_cnts; /* sizeof(CPU_TS_TMR) = 32 bits */

    tmr_val = /* Insert code to read 48-bit down timer value. */ ;
    ts_tmr_cnts = (CPU_TS_TMR)tmr_val; /* Truncate 48-bit timer value to 32-bit timestamp ... */
                                         /* ... timer data type. */ ;
    ts_tmr_cnts = ~ts_tmr_cnts; /* Ones-complement truncated down timer value. */

    return (ts_tmr_cnts);
}
```

## 6-2-6 CPU\_TS\_TmrFreqGet()

Gets CPU timestamp's timer frequency, in Hertz.

### FILES

cpu\_core.h/cpu\_core.c

### PROTOTYPE

```
CPU_TS_TMR_FREQ CPU_TS_TmrFreqGet (CPU_ERR *p_err);
```

### ARGUMENTS

`p_err` Pointer to variable that will receive the return error code from this function:

```
CPU_ERR_NONE  
CPU_ERR_NULL_PTR
```

### RETURNED VALUE

CPU timestamp's timer frequency (in Hertz), if no errors;

0, otherwise.

### REQUIRED CONFIGURATION

Available only if either of the following CPU features is enabled in `cpu_cfg.h`:

- CPU timestamps when either `CPU_CFG_TS_32_EN` or `CPU_CFG_TS_64_EN` is `DEF_ENABLED` (see section 6-1)
- CPU interrupts disabled time measurements when `CPU_CFG_INT_DIS_MEAS_EN` is `#define'd` (see section 7-1)

### NOTES / WARNINGS

None.

---

## EXAMPLE USAGE

```
CPU_TS_TMR_FREQ freq_hz;
CPU_ERR err;

freq_hz = CPU_TS_TmrFreqGet(&err); /* Get CPU timestamp timer frequency. */

if (err == CPU_ERR_NONE) {
    printf("CPU Timestamp Timer Frequency = %d", freq_hz);
} else {
    printf("CPU TIMESTAMP TIMER FREQUENCY NOT AVAILABLE.");
}
```

### 6-2-7 CPU\_TS\_TmrFreqSet()

Sets CPU timestamp's timer frequency, in Hertz.

#### FILES

cpu\_core.h/cpu\_core.c

#### PROTOTYPE

```
void CPU_TS_TmrFreqSet (CPU_TS_TMR_FREQ freq_hz);
```

#### ARGUMENTS

freq\_hz Frequency (in Hertz) to set for CPU timestamp's timer.

#### RETURNED VALUE

None.

## REQUIRED CONFIGURATION

Available only if either of the following CPU features is enabled in `cpu_cfg.h`:

- CPU timestamps when either `CPU_CFG_TS_32_EN` or `CPU_CFG_TS_64_EN` is `DEF_ENABLED` (see section 6-1)
- CPU interrupts disabled time measurements when `CPU_CFG_INT_DIS_MEAS_EN` is `#define'd` (see section 7-1)

## NOTES / WARNINGS

CPU timestamp timer frequency is not required for internal CPU timestamp operations and may optionally be configured by application/BSP initialization functions for use with optional `CPU_TS_to_uSec()` functions to convert CPU timestamps from timer counts into microseconds (see section 6-2-8 “`CPU_TS32_to_uSec()`” on page 42 and section 6-2-9 “`CPU_TS64_to_uSec()`” on page 44).

## EXAMPLE USAGE

```
CPU_TS_TmrFreqSet(2500000u); /* Set CPU timestamp timer frequency to 2.5 MHz. */
```

### 6-2-8 `CPU_TS32_to_uSec()`

Application-defined function to convert a 32-bit CPU timestamp from timer counts to microseconds.

## FILES

`cpu_core.h` / Application's `cpu_bsp.c`

## PROTOTYPE

```
CPU_INT64U CPU_TS32_to_uSec (CPU_TS32 ts_cnts);
```

## ARGUMENTS

`ts_cnts` 32-bit CPU timestamp (in CPU timestamp timer counts).

## RETURNED VALUE

Converted 32-bit CPU timestamp (in microseconds).

## REQUIRED CONFIGURATION

`CPU_TS32_to_uSec()` is an application/BSP function that may be optionally defined by the developer if `CPU_CFG_TS_32_EN` is `DEF_ENABLED` in `cpu_cfg.h` (see section 6-1).

## NOTES / WARNINGS

The amount of time measured by CPU timestamps is calculated by either of the following equations:

$$\text{Time measured} = \text{Number timer counts} * \text{Timer period} * 10^6 \text{ microseconds}$$

$$\text{Time measured} = (\text{Number timer counts} / \text{Timer frequency}) * 10^6 \text{ microseconds}$$

where

Number timer counts	Number of timer counts measured
Timer period	Timer's period in some units of (fractional) seconds
Timer frequency	Timer's frequency in some units of counts per second
Time measured	Amount of time measured, in microseconds

Developer-defined `CPU_TS32_to_uSec()` implementations may convert any number of `CPU_TS32` bits, up to 32, into microseconds.

---

## EXAMPLE TEMPLATE

```
CPU_INT64U CPU_TS32_to_uSec (CPU_TS32 ts_cnts)
{
    CPU_INT64U ts_usec;

    :
    /* Insert code to convert 32-bit CPU timestamp into microseconds. */
    :

    return (ts_usec);
}
```

### 6-2-9 CPU\_TS64\_to\_uSec()

Application-defined function to convert a 64-bit CPU timestamp from timer counts to microseconds.

#### FILES

cpu\_core.h / Application's cpu\_bsp.c

#### PROTOTYPE

```
CPU_INT64U CPU_TS64_to_uSec (CPU_TS64 ts_cnts);
```

#### ARGUMENTS

**ts\_cnts** 64-bit CPU timestamp (in CPU timestamp timer counts).

#### RETURNED VALUE

Converted 64-bit CPU timestamp (in microseconds).

#### REQUIRED CONFIGURATION

CPU\_TS64\_to\_uSec() is an application/BSP function that may be optionally defined by the developer if CPU\_CFG\_TS\_64\_EN is DEF\_ENABLED in cpu\_cfg.h (see section 6-1).

## NOTES / WARNINGS

The amount of time measured by CPU timestamps is calculated by either of the following equations:

$$\text{Time measured} = \text{Number timer counts} * \text{Timer period} * 10^6 \text{ microseconds}$$

$$\text{Time measured} = (\text{Number timer counts} / \text{Timer frequency}) * 10^6 \text{ microseconds}$$

where

Number timer counts	Number of timer counts measured
Timer period	Timer's period in some units of (fractional) seconds
Timer frequency	Timer's frequency in some units of counts per second
Time measured	Amount of time measured, in microseconds

Developer-defined `CPU_TS64_to_uSec()` implementations may convert any number of `CPU_TS64` bits, up to 64, into microseconds.

## EXAMPLE TEMPLATE

```
CPU_INT64U CPU_TS64_to_uSec (CPU_TS64 ts_cnts)
{
    CPU_INT64U ts_usec;

    :
    /* Insert code to convert 64-bit CPU timestamp into microseconds. */
    :

    return (ts_usec);
}
```

# Chapter 7

## **μC/CPU Interrupts Disabled Time Measurement**

μC/CPU can measure the maximum amount of time interrupts are disabled during calls to `CPU_CRITICAL_ENTER()` / `CPU_CRITICAL_EXIT()` is measured and saved. There are two maximum interrupts disable time measurements, one resetable and the other non-resetable, both measured in units of CPU timestamp timer counts.

The interrupts disabled time measurement feature is available only if `CPU_CFG_INT_DIS_MEAS_EN` is `DEF_ENABLED` in `cpu_cfg.h`. Note that this feature requires that the application/developer provide CPU timestamp timer functions (see section 6-2-4 “`CPU_TS_TmrInit()`” on page 35 and section 6-2-5 “`CPU_TS_TmrRd()`” on page 37).

### **7-1 μC/CPU INTERRUPTS DISABLED TIME MEASUREMENT CONFIGURATION**

The following μC/CPU interrupts disabled time measurement configurations must be configured in `cpu_cfg.h`:

`CPU_CFG_INT_DIS_MEAS_EN`

Includes code to measure and return maximum interrupts disabled time. This feature is enabled if the macro is `#define'd` in `cpu_cfg.h`.

`CPU_CFG_INT_DIS_MEAS_OVRHD_NBR`

Configures the number of times to measure and calculate the interrupts disabled time measurement overhead.

## **7-2 µC/CPU INTERRUPTS DISABLED TIME MEASUREMENT FUNCTIONS**

### **7-2-1 CPU\_IntDisMeasMaxGet()**

Gets (non-resetable) maximum interrupts disabled time.

#### **FILES**

cpu\_core.h/cpu\_core.c

#### **PROTOTYPE**

```
CPU_TS_TMR CPU_IntDisMeasMaxGet (void);
```

#### **ARGUMENTS**

None.

#### **RETURNED VALUE**

(Non-resetable) maximum interrupts disabled time (in CPU timestamp timer counts).

#### **REQUIRED CONFIGURATION**

Available only if CPU\_CFG\_INT\_DIS\_MEAS\_EN is #define'd in cpu\_cfg.h (see section 7-1).

#### **NOTES / WARNINGS**

None.

#### **EXAMPLE USAGE**

```
CPU_TS_TMR time_max_cnts;  
  
time_max_cnts = CPU_IntDisMeasMaxGet(); /* Get maximum interrupts disabled time. */
```

## **7-2-2 CPU\_IntDisMeasMaxCurGet()**

Gets current/resetable maximum interrupts disabled time.

### **FILES**

cpu\_core.h/cpu\_core.c

### **PROTOTYPE**

```
CPU_TS_TMR CPU_IntDisMeasMaxCurGet (void);
```

### **ARGUMENTS**

None.

### **RETURNED VALUE**

Current maximum interrupts disabled time (in CPU timestamp timer counts).

### **REQUIRED CONFIGURATION**

Available only if CPU\_CFG\_INT\_DIS\_MEAS\_EN is #define'd in cpu\_cfg.h (see section 7-1).

### **NOTES / WARNINGS**

None.

### **EXAMPLE USAGE**

```
CPU_TS_TMR time_max_cnts;  
  
time_max_cnts = CPU_IntDisMeasMaxCurGet(); /* Get current maximum interrupts disabled time. */
```

### **7-2-3 CPU\_IntDisMeasMaxCurReset ()**

Resets current maximum interrupts disabled time.

#### **FILES**

cpu\_core.h/cpu\_core.c

#### **PROTOTYPE**

```
CPU_TS_TMR CPU_IntDisMeasMaxCurReset (void);
```

#### **ARGUMENTS**

None.

#### **RETURNED VALUE**

Maximum interrupts disabled time (in CPU timestamp timer counts) before resetting.

#### **REQUIRED CONFIGURATION**

Available only if `CPU_CFG_INT_DIS_MEAS_EN` is `#define'd` in `cpu_cfg.h` (see section 7-1).

#### **NOTES / WARNINGS**

None.

#### **EXAMPLE USAGE**

```
CPU_TS_TMR time_max_cnts;  
  
time_max_cnts = CPU_IntDisMeasMaxCurReset(); /* Reset current maximum interrupts disabled time. */
```

## Appendix

# A

## $\mu$ C/CPU Licensing Policy

You need to obtain an “Object Code Distribution License” to embed  $\mu$ C/CPU in a product that is sold with the intent to make a profit. Each individual product (*i.e.*, your product) requires its own license, but the license allows you to distribute an unlimited number of units for the life of your product. Please indicate the processor type(s) (*i.e.*, ARM7, ARM9, MCF5272, MicroBlaze, Nios II, PPC, *etc.*) that you intend to use.

For licensing details, contact us at:

Micrium  
1290 Weston Road, Suite 306  
Weston, FL 33326  
USA

Phone: +1 954 217 2036  
Fax: +1 954 217 2037  
E-mail: [Licensing@Micrium.com](mailto:Licensing@Micrium.com)  
Web: [www.Micrium.com](http://www.Micrium.com)